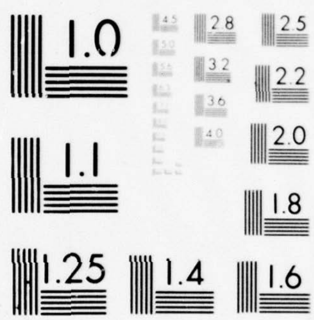
AD-A044 438 ROCHESTER UNIV NY DEPT OF COMPUTER SCIENCE F/G 9/2
RPL, A BCPL-BASED PRODUCTION LANGUAGE SYSTEM,(U)
APR 77 K A LANTZ, C B MELTZER N00014-75-C-1091
UNCLASSIFIED TR-15 NL

1 OF 1
AD
A044438

END
DATE
FILMED
10-77
DDC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

電腦科學

B.S.

Department of Computer Science
University of Rochester
Rochester, New York 14627

11

6

RPL, A BCPL-Based
Production Language System

by

10

Keith A./Lantz
Clifford B./Meltzer

14 TR15

Computer Science Department
The University of Rochester
Rochester, NY 14627

11 April 1977

12 56p.

This research was supported in part by the Alfred P. Sloan Foundation under Grant #74-12-5; in part by the National Science Foundation under Grant #MCS76-10825; and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by ONR under Contract #N00014-75-C-1091;

15 NSF-MCS-76-10825

410 386

Contents

Preface

This report is intended to serve two purposes. First, it should provide the interested reader with what is thought to be a particularly elegant design for a production language. To this end, the Rochester Production Language is discussed, as far as possible, in implementation-independent terms. To provide the reader with a feel for the actual use of production language, salient points of the RPL Translator Writing System are also discussed; in particular, the interface to user-supplied BCPL semantic routines. The reader is expected to have a rudimentary knowledge of production language (suggested reading is [Gries]).

Second, this report is meant to serve as a reference manual for the Rochester Production Language. It is not, however, a detailed user's guide to the RPL Translator Writing System [Lantz].

The syntactic notation used here is a variant of BNF. Braces { and } are metasymbols used to represent repetition and/or alternation. Some possible constructs are:

{<x>}*	zero or more occurrences of <x>
{<x>}+	one or more occurrences of <x>
{<x>}n	0 to n occurrences of <x>
{<x> ¦ <y>}	either <x> or <y>

Brackets [and] indicate an optional string.

(The syntactic metasymbols < and > are also RPL source symbols. Their use in a particular instance is disambiguated by the fact that RPL reserved words are uppercase (e.g., <RPL>), whereas BNF nonterminals are lower case (e.g., <program>).)

I. Overview of the RPL System

In designing a translator writing system, two meta-languages are required: 1) the meta-language for describing the syntax of target programming languages; 2) the meta-language for writing the semantic routines for the target languages. Production language is a meta-language of the first type, specifying the operation of the recognizer for the target language. It provides convenient primitives for the deterministic description of the target syntax; it is readable; and it is machine-independent. Hence the motivation behind the Rochester Production Language.

The RPL Translator Writing System was developed primarily as an aid in the generation of parsers for interactive command languages. It was therefore natural to choose as the semantic meta-language a language already in use for system development; namely, BCPL [Curry].

The basic form of the RPL System is as given in Figure 1. Provided with an RPL source file defining the syntax of a target language, the RPL compiler generates tables for the target parser. Currently, these tables consist of BCPL source files, which are then compiled and loaded with the RPL interpreter. Also loaded with the interpreter are the BCPL semantic routines for the target language. This set of load modules -- compiler-generated "tables", interpreter, and semantic routines -- comprises the translator for the target language.

We have then a table-driven translator based on a recognizer using a single stack. Each element of the stack consists of a syntactic construct and its associated semantics. As tokens are scanned from the target source file they are placed on the stack. Semantic routines are called as the syntactic constructs with which they are associated are recognized. Depending on the nature of these semantic routines, the translator may act as a compiler (generating object code), or as an interpreter (directly executing the target source file).

II. The Rochester Production Language

1. Reserved Words, Identifiers, and Constants

RPL reserved words consist of pseudo-op "block identifiers" -- e.g., <RPL>, <SCANNER>, <CLASSES>, <END> -- and "operational" reserved words -- e.g., INTEGER, EXEC, SCAN. Block identifiers delimit the three major parts of the RPL program as well as sub-blocks thereof; they begin with a < and end with a >. All reserved words are restricted to upper case.

RPL identifiers consist of nonterminals, aliases, class names, labels, and semantic-routine names. In general, these consist of sequences of letters, digits, and -, starting with a letter. The block markers < and > are available for use as head and trail characters, respectively; they are particularly useful for representing nonterminals. The @ is also provided as a possible head character, should the user wish to distinguish, for example, all class names. Identifiers are not restricted to upper case, and may be up to 255 characters long.

RPL recognizes the following constructs as constants:

A string of digits is interpreted as a decimal integer, not to exceed 32767.

A $ followed by any printing character other than the escape * represents a constant whose value is the 7-bit ASCII code of the character.

A sequence of printing characters (including space) enclosed in double quotes is a string constant. A string has maximum length 255. As with character constants, * is an escape.

The following escapes are recognized:

*b *B	backspace
*c *C	carriage return
*d *D	delete
*e *E	escape
*f *F	form feed
*l *L	line feed
*s *S	space
*t *T	tab
*"	"
**	*
*nnn	3-digit octal number "nnn"

(Note that character and string constants need be used only where target-language symbols conflict with RPL parsing rules or contain non-printing characters -- such as space in "GO TO" or

carriage return in $*C. Such a constant is, therefore, always interpreted as a target-language symbol (usually a delimiter or reserved word). See Section II-3.)

2. <RPL> ... The RPL Program ... <ENDRPL>

The ultimate aim of an RPL program is to specify a sequence of productions defining the actions of the target parser. But we must first specify the production alphabet, both terminals and nonterminals. To this end, an RPL program is divided into three main parts: 1) the scanner constructor; 2) RPL declarations; 3) the program body -- i.e., the productions. Each of these is considered separately. (Technically both the scanner constructor and declarations are optional. In the case of the former, certain defaults are provided.)

3. <SCANNER> ... The Scanner Constructor ... <ENDSCANNER>

The RPL scanner is intended for use with any programming language as input (indeed, it is used for parsing RPL itself). Being table-driven, it is necessary for the RPL compiler to generate tables which describe the terminal symbols of the target language. This is effected via the character-class and reserved symbol "declarations" of the scanner constructor.

3.1 Character Classes

A character-class table is used to find the "class" or type of the character being scanned. There are seven character classes, some of which may be observed to be mutually exclusive:

<DIGIT>	- digits (default: 0 - 9)
<HEAD>	- first characters of an identifier (default: A - Z, a - z)
<TRAIL>	- other-than-first characters of an identifier (default: <HEAD> characters)
<STRINGCONSTANT>	- string constant "delimiters" (default: ")
<CHARACTERCONSTANT>	- character constant "delimiters" (no default examples: $)
<ESCAPE>	- escape characters (no default examples: *)
<INVISIBLE>	- invisible terminators (default: $*S, $*T)

Character classes are specified by simply listing the member characters separated by RPL invisible terminators -- space, tab, and carriage return:

```
<char-class> {<target-char>}* <END>
```

<char-class> is one of the character-classes listed above. <target-char> is a printing character, an RPL character-constant, or a single-character RPL string-constant -- e.g., A, $*t, "*C".

A null character-class declaration -- <char-class> <END> -- would serve to override the defaults such that no character belonged to the given class.

3.2 <DELIMITERS> {<delimiter>}+ <END>

A <delimiter> is either a <target-char> (II-3.1), in the case of a single-delimiter, or a 2-character printable symbol or RPL string-constant (e.g., <=, "^*C"), in the case of a double-delimiter.

Delimiters are target reserved symbols and hence must be stored in the dictionary for the target parser. However, their first characters also implicitly specify two other character classes:

delimiter	- single-character delimiters (default: $*C others: -, *, or $*177)
double-delimiter	- double-character delimiters; the first character is a member of the character-class (no default examples: **, <=, or "^*C" where *, <, and ^, respectively, are implicitly specified as members of the class)

Longer delimiters can be specified as target reserved words, but only if their constituent characters correspond to the rules for target identifiers. For example, if .LE. is thought of as a delimiter rather than a reserved word, then . must be a <HEAD> and L, E, and . must be <TRAIL>s in order for .LE. to appear among the <RESERVEDWORDS>. In this sense, we have effectively restricted delimiters to two characters.

3.3 <RESERVEDWORDS> {<reserved-word>}+ <END>

<reserved-word>s are target identifiers according to the character classes <HEAD> and <TRAIL>. Therefore, the declarations, if they exist, for <HEAD> and <TRAIL> characters must precede the <RESERVEDWORDS> declaration. As with the character class declarations, <reserved-words>s are separated by space, tab, and carriage return.

When a target reserved word conflicts with RPL parsing rules it must be specified as an RPL string or character constant. For example, if <END> is a target reserved word and we are parsing

the <RESERVEDWORDS> declaration, then the target <END> must be specified as "<END>" in order not to conflict with the RPL <END> which terminates the declaration. String constants are also used when the reserved word contains non-printing characters -- such as the space in "GO TO".

When string and character constants are used, the target reserved word can later be assigned an RPL alias (see II-4.2) or simply specified as that same constant -- "GO TO" -- wherever it occurs in the program.

3.4 <COMMENTS> {<target-symbol> <target-symbol>}+ <END>

Each <target-symbol> pair specifies a start-of-comment delimiter and its associated end-of-comment delimiter -- e.g., COMMENT and $;, or /* and */. Each such <target-symbol> must have previously been specified in the <DELIMITERS> or <RESERVEDWORDS> declaration(s) (II-3.2 and II-3.3).

3.5 Example

This example and those to follow deal with a variant of ALGOL that we shall call SIMPLEGOL. The full BNF syntax for SIMPLEGOL can be found in Appendix B.

The scanner for SIMPLEGOL might be specified as follows:

```
<SCANNER>

  <HEAD>  A B C D E F G H I J K L M
          N O P Q R S T U V W X Y Z  <END>

  <TRAIL> A B C D E F G H I J K L M
          N O P Q R S T U V W X Y Z _   <END>

  <INVISIBLE>   $*s  "*T"      <END>

    /* <DIGIT>s are 0 - 9                                  */
    /* <STRINGCONSTANT> is "                               */
    /* There is no <CHARACTERCONSTANT>                     */
    /* There is no <ESCAPE>                                */

  <DELIMITERS>  , :  ;  (  )  +  -  *  /  ~  <  >  =
                :=  <=  >=  ~=  <END>

  <RESERVEDWORDS>

        AND  ARRAY
        BEGIN  BREAKOUT
        CASE  COMMENT
        DEFAULT  DO
        ELSE  END
        FI  FOR  FROM  FUNCTION
        IF  INTEGER
        NEXT
        OD  OF  OR
        REAL  REFERENCE  RETURN
        SKIPREST  STEP
        THEN  TO
        UNTIL
        VALUE  VECTOR
        WHILE  WITH

  <END>

  <COMMENTS>   COMMENT  ;      <END>

<ENDSCANNER>
```

4. <DECLARATIONS> ... RPL Declarations ... <ENDDECLARATIONS>

RPL declarations consist of nonterminals, aliases, constants, and (RPL) classes. Classes must follow nonterminals and aliases. No other restriction is placed on the order of declarations.

4.1 <NONTERMINALS> {<nonterminal>}+ <END>

<nonterminal>s are RPL identifiers. They can most easily be thought of as equivalent to the nonterminals in a BNF description of the target language. Hence, they are tokens the user may want to put on the stack as "markers" of partially completed productions.

4.2 <ALIASES> {<target-symbol> <alias>}+ <END>

<target-symbol>s must adhere to the syntactic rules given above for the scanner declarations. That is, they must be target language delimiters or reserved words, possibly expressed as string or character constants. <alias>es may be RPL identifiers or other <target-symbol>s.

<alias>es are provided for three reasons. First, an <alias> in the form of an RPL identifier should be provided for any <target-symbol> (delimiter or reserved word) which conflicts with RPL parsing rules -- e.g. CONTINUE for "^*C" or GOTO for "GO TO". Such an <alias> should also be provided for target symbols which are duplicates of RPL reserved symbols -- e.g., INT for INTEGER, or COLON for :.

Second, RPL (and hence BCPL) identifier <alias>es must be provided for all those members (which are not already identifiers) of RPL classes which are arguments to semantic routines. This allows the RPL compiler to generate meaningful manifest constants for use by the BCPL semantic routines. (See II-4.3 and II-5.4.2)

Lastly, this scheme provides the means for equivalencing target symbols -- e.g., <= and "le" -- which have previously been declared. In this case the <alias> is also a <target-symbol>. Either symbol appearing in the target source file will be interpreted as the same token. (It is, of course, possible to extend this to more than 2 symbols via multiple <alias>es -- e.g., <=, "le", and "leq". This could be done as follows:

```
<ALIASES> <= le  le leq  <END>
```

Aliasing is also convenient for abbreviations -- e.g., PROCEDURE

and PROC.)

4.3 <CLASSES> {<class> {<class-member>}+ }+ <END>

A <class> is a conventional RPL identifier. Classes are specified on a one-class-per-line basis (with continuation). The <class-member>s are, in general, target reserved symbols, aliases, and nonterminals. If a <class-member> is a formerly specified <class>, all members of that previously specified class become members of the current class as well. A <class-member> may also be one of the basic metasymbols I, INTEGER, STRINGCONST, or CHARCONST (see II-5.1); or one of the predeclared classes RESERVEDWORDS, consisting of all target reserved words, or DELIMITERS, consisting of all target delimiters.

Generic classes of symbols are often useful in the interest of efficiency and simplicity; they make it possible to replace many similar productions with one production. For example, assume that we have a target language which can deal with relational expressions:

```
<relational>  ::=  <term>  {<relop>  <term>}*
<relop>       ::=  lt  |  le  |  eq  |  ne  |  ge  |  gt
```

This might naturally lead to an RPL program containing the 6 productions:

```
<term> lt <term> ANY    /* ... actions ... */
          ...
<term> gt <term> ANY    /* ... actions ... */
```

It is possible, however, to define a <class> for the relational operators:

```
<CLASSES>  @relop  lt le eq ne ge gt <END>
```

and replace the 6 productions with a single production:

```
<term> @relop <term> ANY     /* ... actions ... */
```

Typical actions here are SWITCH (see II-5.4.7) or EXEC (see II-5.4.2).

(See Sections II-4.5 and II-5 for further details.)

4.4 <CONSTANTS> {<constant> <integer>}+ <END>

A <constant> is an RPL identifier, and is assigned the associated <integer> value. <constant>s may be used within the RPL productions wherever an integer may be used (see II-5.4).

<constant>s are particularly useful for interfacing with the semantic routines. As will be seen in section II-5.4.2, an integer may be passed as a parameter to a semantic routine; in the interest of readability and reliability, each such parameter should have a name common to both the RPL program and the semantic routines. Moreover, a semantic routine can return a result to the recognizer via the SemSwitch variable (see II-5.4.8 and III-5.1). As with integer parameters, each such result should have a name. <constant>s can also be used in the place of integer ERROR and HALT codes. For these reasons, the RPL compiler generates BCPL manifest declarations for the <constant>s to be used by the semantic routines.

4.5 Example

RPL declarations for SIMPLEGOL might look as follows:

```
<DECLARATIONS>

  <NONTERMINALS>

       <program>  <expr>
       <conditional>
       <loop>  <prologue>  <epilogue>
       <assignment>
       <case-expr>  <case-body>  <case>  <case-label-list>
                    <case-label>
       <arithmetic>  <conjunction>  <relational>  <term>
                     <factor>  <primary>
       <compound>  <decl>  <id-list>
                   <function>  <formals>  <f-decl>  <f-type>
                   <body>
       <control>
       <variable>  <subscriptlist>
       <begin>  "<end>"
       /* quotes required to distinguish from the <END>
          terminating the <NONTERMINALS> */

  <END>

  <ALIASES>

          /* delimiters and reserved words which may otherwise
             cause problems since they conflict with RPL
             reserved symbols                              */

       : colon  , comma  ( lp      ) rp     := assign
       DEFAULT defalt

          /* operators                                     */

       = eq     ~= ne    <= le    >= ge     < lt     > gt
       + plus   - minus
       * mult   / div
       ~ not

  <END>

  <CLASSES>

       @BooleanTest    WHILE  UNTIL
       @ForTest        TO  @BooleanTest
       @LoopControl    SKIPREST  BREAKOUT

       @Type           REAL  INTEGER
       @CallType       REFERENCE  VALUE
```

```
            /* operators                              */

         @RelOp           =  ~=  <=  >=  <  >
         @AddOp           $+  minus
         @MultOp          "**"  /
         @PrefixOp        -  ~

    <END>

  <ENDDECLARATIONS>
```

5. <PRODUCTIONS> ... RPL Productions ... <ENDPRODUCTIONS>

The "heart" of a RPL program is a sequence of productions, each of which has the form:

{<label>:}* <left-part> [=> <right-part>] {<actions>}*

Note that the minimal production consists simply of a <left-part>.

Productions are line-oriented -- i.e., terminated by a carriage return. Continuation is provided via the ^ pseudo-op. Multiple labels may, however, be written on separate lines without continuation.

The <label> is an optional production "name." All labels must be unique RPL identifiers, or the global labels START, SCANERROR, SWITCHERROR, and SEMSWITCHERROR. START specifies the first production to be interpreted, and defaults to the first production in the RPL source file; for the error labels see III.5. The colon must be present for a label to be interpreted as such, and not as an element of the <left-part>.

<left-part> is a list of target symbols, nonterminals, aliases, classes, and RPL metasymbols, which is to be matched against the current top elements of the RPL stack.

<right-part> is a (optional) list of similar symbols which will replace the <left-part> on the stack if the production "succeeds" -- i.e., if the <left-part> matches the top of the stack.

The <action>s, if any, are to be executed after the <left-part> has matched and the stack transformation has been made.

5.1 Metasymbols

The RPL metasymbols are:

I	- match a target identifier
INTEGER	- match a target number (integer)
STRINGCONST	- match a target string constant
CHARCONST	- match a target character constant
ANY	- match anything

In order for a metasymbol to appear in a <right-part> it must have appeared in the corresponding <left-part>. If a metasymbol does occur in the <right-part>, the symbol which was matched by that metasymbol in the <left-part> will appear on the transformed stack.

5.2 Classes

When used in a <left-part>, an RPL-class matches any symbols with which it has been associated in an RPL-class declaration (see II-4.3). If the class then appears in the <right-part>, the member of the class which was actually matched is the symbol to be pushed onto the stack.

5.3 Multiplicity of Like Symbols

Assume we encounter productions of the following type:

```
ANY Y ANY => ANY U ANY V          /* ... actions ... */
@addop T @addop => T @addop       /* ... actions ... */
```

In the first case, which ANY in the <right-part> receives the semantics of the leftmost ANY in the <left-part>? In the second case, whose semantics does the @addop in the <right-part> receive -- those of the left or of the right @addop in the <left-part>?

Two ways to solve these problems are provided. First, there is a provision for indexing of any RPL "symbols" in the left and right-parts -- i.e., all nonterminals, classes, and metasymbols. Any such variable may be indexed by 1, 2, or 3 such that multiple occurrences may be uniquely identified. For example, given:

```
ANY1 Y ANY2 => ANY1 U ANY1 V
```

when the top of stack looks like:

```
top -  X
       Y
       W
```

then the stack would be transformed to:

```
top -  V
       W
       U
       W
```

On the other hand, given:

```
ANY1 Y ANY2 => ANY2 U ANY1 V
```

with the same stack, yields:

```
top -  V
       W
       U
       X
```

If all multiplicities are not resolved in the <left-part>, there are two cases to consider. If the conflicting symbols are not indexed, a canonical ordering is assumed. First, there must be at least as many occurrences of the symbol in question in the <left-part> as in the <right-part>. Then, counting right to left, the i-th occurrence of the symbol in the <right-part> becomes the token currently on the stack which matched the i-th occurrence of the symbol in the <left-part>. Hence, given:

```
ANY Y ANY => ANY U ANY V
```

and the initial stack as above, the transformed stack would be:

```
top -  V
       X
       U
       W
```

which is the same as if we had written:

```
ANY1 Y ANY2 => ANY1 U ANY2 V
```

However, if multiplicities occur in the <left-part> due to identical indexing, it is assumed that the user wishes the same symbol to be matched in each instance. The rightmost occurrence of the multi-variable (i.e., the one at the top of the stack) is taken as the symbol to be matched by the remaining occurrences. Hence, given:

```
ANY1 Y ANY1 => ANY1 U
```

and the same initial stack the production would fail -- the left ANY1 would not match the X matched by the right ANY1.

Although the examples have dealt with the metasymbol ANY, the same ideas apply to any other metasymbol, nonterminal, or class name.

5.4 RPL Actions

Actions can be divided into four types:

```
I / O         - SCAN

semantic      - EXEC

control       - SGOTO
                FGOTO
                CALL
                RETURN
                SWITCH
                SEMSWITCH
```

exception - ERROR
HALT
CLEAN

Note that the specified actions, with the exception of SGOTO and FGOTO, are performed sequentially in the order in which they are written in the production.

5.4.1 SCAN [<number>]

SCAN specifies that the next target symbol is to be scanned and pushed onto the stack. SCAN <number> specifies that the next <number> symbols are to be so scanned. <number> is an integer or RPL <constant>.

5.4.2 EXEC {<proc-name> [({<class> ¦ <number>})]}+

EXEC specifies that a list of semantic routines is to be called. The <proc-name>s given in this list should be the names of user-supplied (BCPL) semantic routines. EXEC accepts either a class or a number as a parameter to a semantic routine. In the latter case, the integer itself may be given, or a declared <CONSTANT> (II-4.4) may be specified.

In the case of a class parameter, the class must have occurred in the <left-part> of the production. The value passed to the semantic routine is the token number of the symbol (on the stack) which actually matched the class. This, of course, presents a problem: Since token numbers are generated by the RPL compiler, the user has no immediate means of knowing which token number corresponds to which symbol. Therefore, the compiler generates manifests (macro constants) for all those tokens which are members of classes used as parameters to semantic routines. For example, for the class @relop, the manifests EQ, NE, LT, LE, GT, and GE would be generated, their values being the token numbers generated by the RPL compiler (see II-4.2).

Semantic routines manipulate the stack based on the stack contents immediately before and after the stack transformation (if any) of the current production. They should therefore be called prior to performing any other actions which might change the stack. Specifically, it is usually unwise to perform a SCAN or CALL prior to an EXEC.

5.4.3 SGOTO <label>

SGOTO specifies the production to which control is passed should the current production succeed. Control is transferred after all other actions are performed. Hence, SGOTO may appear anywhere in the action sequence. SGOTO defaults to the next

production.

5.4.4 FGOTO <label>

FGOTO specifies the production to which control is passed should the current production fail. Control is transferred immediately after it is discovered that the <left-part> doesn't match. FGOTO defaults to the next production.

5.4.5 CALL <label>

CALL is an RPL-subroutine call. That is, it is a "push-jump" to a production where execution will continue until a matching RETURN is encountered. The environment of the production from which the CALL is made is not preserved -- i.e., a RETURN does not reinitialize the stack.

5.4.6 RETURN

RETURN causes a "pop-jump" to the point of the last executed call.

5.4.7 SWITCH <class> {(<case-list> : <label>)}+

A <case-list> is a non-empty list consisting of members of the <class> or the reserved word DEFAULT. The <class> must have occurred in the <left-part> of the production; it is evaluated to the class-member which was matched. If the <class> appears more than once in the <left-part>, the evaluation is based on the rightmost occurrence (see II-5.3).

The effect of the SWITCH action is as follows: If a <case-list> contains a case equal to the evaluated <class>, a jump is made to the production specified by the corresponding <label>. If no <case> matches the evaluated <class>, a jump is made to the DEFAULT production, if there is one; if, in this case, no DEFAULT exists, a run-time error will occur.

The (<case-list>:<label>) pairs are allowed to occur in any order. Only one DEFAULT is allowed over all <case-list>s.

5.4.8 SEMSWITCH {(<semcase-list> : <label>)}+

A <semcase-list> is a non-empty list consisting of integers, <constants> (II-4.4), or the reserved word DEFAULT. SEMSWITCH "switches" on the global (to the target translator) variable SemSwitch, which is setable by the user's semantic routines. SEMSWITCH allows the user to specify different control paths dependent on the result of a semantic routine -- e.g., if a

semantic error occurs.

The interpretation of a SEMSWITCH parallels that of SWITCH.

5.4.9 ERROR <number>

ERROR specifies that the user error routine UserError is to be called with <number> as a parameter -- an index to an error message, perhaps. <number> is an integer or an RPL <constant>.

5.4.10 HALT <number>

HALT specifies that the RPL interpreter is to terminate program execution. The user "halt" routine UserHalt is called with <number> as a parameter. <number> is an integer or an RPL <constant>.

5.4.11 CLEAN

CLEAN is used for drastic error recovery. It empties the production stack(s), flushes the current input line, and generally re-initializes the world. Typically, prior to performing a CLEAN, the user wil call a semantic routine to perform any necessary semantic error recovery. A typical follow-up to CLEAN is to "restart" the interpreter at the first (START) production.

5.5 Example

See the example in Appendix C. The interested reader may, however, wish to write the productions for SIMPLEGOL (see Sections II-3.5 and II-4.5, as well as Appendix B).

6. RPL Source File Conventions

We reiterate here that only RPL-class declarations and productions are line-oriented. In these instances continuation is provided via the ^ pseudo-op. The ^ and the remainder of the line in which it occurs is ignored. In all other declarations a carriage return is treated as an invisible terminator -- i.e., a space.

Comments may appear anywhere in the source text. They begin with /* and end with */. The comment-delimiters and all intervening text, independent of line boundaries, are ignored.

III. The RPL Translator-Writing System

1. Overview

The RPL System consists of: 1) the RPL table-driven scanner; 2) the RPL compiler; 3) the RPL interpreter. When the scanner and interpreter are combined with semantic routines and the compiler-generated tables for a particular target language, a translator for the target language results. The details of the RPL system are outlined in Figure 2.

2. The Compiler

The functions of both the RPL scanner and compiler are implicitly outlined in the discussion of the Rochester Production Language (Section II). We reiterate here that the compiler receives as input an RPL source program defining the recognizer for the target language. The compiler produces three BCPL source files as output:

1) a header file consisting of:
   a) external declarations for the semantic routines
   b) manifests for the tokens used in the semantic routines: integer parameters; token numbers for the members of a class parameter; and SemSwitch results

   This file is included (via a BCPL "get") in both the target initialization file (see the following) and the semantic routine file. (See Appendix E for an example.)

2) a file containing initialization routines for:
   a) the character-class table and dictionary for the scanner
   b) for each target language symbol, a vector of the RPL classes to which it belongs
   c) the production table
   d) the vector of semantic routines

   This file is compiled and loaded with the RPL interpreter.

3) a debugger initialization file (not to be discussed here)

3. The Interpreter

The RPL Interpreter has two phases:

1) initialize the target environment via the initialization routines mentioned in III.2

2) interpret a target source file

The second phase is the focus of the remainder of this document.

The interpreter always starts at the first (START) production. Each production specifies a pattern matching and possible stack transformation. The basic outline of the interpretation process is given in Figure 3.

The interpreter tries to match the <left-part> of the current production against the top of the (syntactic) stack. The rightmost symbol in the <left-part> must match the top of the stack, the second-from-the-right must match the second element on the stack, and so on. If a <left-part>-symbol fails to match its respective symbol on the stack, then the pattern-match fails, in which case we proceed to the production specified by the FGOTO (failure-goto) of the current production (which defaults to the next production). If no match exists between any production and the stack, the interpreter is aborted.

When, however, a match occurs, the interpreter will perform the actions specified. If a stack transformation is specified, the <left-part> is popped off the stack; if the <right-part> is non-empty, it is pushed on the stack, the leftmost <right-part> symbol being stacked first. Having executed all actions in the order specified (aside from FGOTOs and SGOTOs), the interpreter will proceed to the production specified by the SGOTO (success-goto, which defaults to the next production). The basic operation of each action should be obvious from the discussion in Section II-5.4.

4. Semantic Routines

We remark again that the stack used by the interpreter comprises two substacks, one for syntax and one for semantics. That is, each element of the stack contains a syntactic construct -- a target language symbol or nonterminal -- and the "meaning" or semantics of that construct. When the scanner pushes a delimiter or reserved word onto the stack, the semantics of that symbol are effectively undefined. However, the semantics for a metasymbol is defined as the particular entity -- identifier, integer, string constant, or character constant -- to which the metasymbol refers:

I -- a pointer to the identifier in string space (the user is responsible for entering the identifier in his own dictionary or symbol table, and, perhaps, changing the semantics to a reference to the table entry)

INTEGER -- the integer itself

STRINGCONST -- a pointer to the string in string space

CHARCONST -- the ASCII code for the character

Semantics are manipulated by user-supplied semantic routines. Such manipulation is the only means by which delimiters (such as operators) and reserved words obtain meaningful semantics. For example, the nonterminal <arithmetic> will eventually contain the semantics of an arithmetic expression; namely, the value of the expression.

The semantic routines must therefore have access to the (semantic) stack. To this end the interpreter sets the variables L1, L2, ... L10 to refer to the top, second, ..., tenth semantic elements of the stack, respectively, considering the stack before the stack transformation, if any, has been made. L1, ..., L10 thus refer to the semantics of the <left-part>. The variables R1, R2, ..., R10, on ther other hand, refer to the semantic elments of the stack considering the stack configuration after the stack transformation has been made. They thus refer to the semantics of the <right-part>. For example, given:

```
a  b  c  =>  d  e  f
L3 L2 L1     R3 R2 R1
```

L1 will contain the semantics of c; L2 contains those of b; and L3 contains those of a. R1 contains the semantics of f, etc.

A semantic routine, then, will usually manipulate the R's. For example, consider the production:

```
<factor> + <factor> ANY => <factor> ANY  EXEC Add
```

Assume the stack appears as follows when the production is reached:

```
                syntax         semantics

       top -    [anything]     [any-semantics]
                <factor>       12
                +              [no semantics]
                <factor>       8
```

Then on entry to the semantic routine Add we would have the following values for L's and R's:

```
       L1 = [any-semantics]              R1 = [any-semantics]
       L2 = 12                           R2 = 12
       L3 = [no semantics]
       L4 = 8
```

Within Add the following statement would no doubt appear:

```
                R2 = L2 + L4
```

thus changing the semantics of the <factor> in the right-part. Return from Add yields the final stack:

```
                syntax         semantics

       top -    [anything]     [any-semantics]
                <factor>       20
```

It should be obvious from this discussion that semantic routines should be executed (that is, all EXECS should be performed) prior to performing any other action that might affect the stack -- i.e., SCAN or CALL. (See II-5.4.2)

5. Run-Time Error Handling

There are basically two classes of run-time errors, those resulting from errors in the target source file and those resulting from an incorrectly specified recognizer.

5.1 Target Language Errors

Lexical (scanner) errors are fielded in part by the RPL scanner. Some action is, however, necessary in the recognizer itself. That is, when a production requests a SCAN, what does the interpreter do when the SCAN fails? By default, the current input line is flushed, the stack is emptied, and the interpreter is restarted at the first (START) production. This being somewhat drastic, an alternative is provided. A single production may be labeled as the SCANERROR production, to which the interpreter will proceed whenever a scanner error occurs. The sequence of productions following the SCANERROR will determine error recovery. Note that such a branch is not a CALL; the interpreter cannot return to the point of error (in the middle of a production) and continue.

Syntactic (parser) errors are fielded via the ERROR and HALT constructs (II-5.4.9 and II-5.4.10). The user supplies the corresponding UserError and UserHalt routines along with his semantic routines.

Semantic errors may be fielded via the SEMSWITCH construct, perhaps in conjunction with ERROR and HALT. Any semantic routine may set the value of the global variable SemSwitch. Upon return from a routine, the interpreter may then branch on this value (via the SEMSWITCH construct) to other productions. Some of these productions may specify, at RPL-source level, semantic error posting and recovery.

5.2 Recognizer Errors

If the recognizer has been incorrectly (or incompletely) specified, several errors may occur. First, it might happen that no production matches the current top of stack. In this case the interpreter runs off the bottom of the production table and aborts. A similar fate awaits the last production in the RPL program, which may not specify any GOTOs.

More commonly, SWITCHes and SEMSWITCHes may have been incompletely specified. That is, all possible cases may not have been accounted for; this, of course, means that DEFAULT was not used. In these instances two global labels similar to SCANERROR are provided: SWITCHERROR and SEMSWITCHERROR. The action taken

by the interpreter parallels that taken for a scanner error.

6. The Debugger

An interactive debugger is also provided as part of the RPL interpreter. It is discussed in the RPL User's Guide [Lantz].

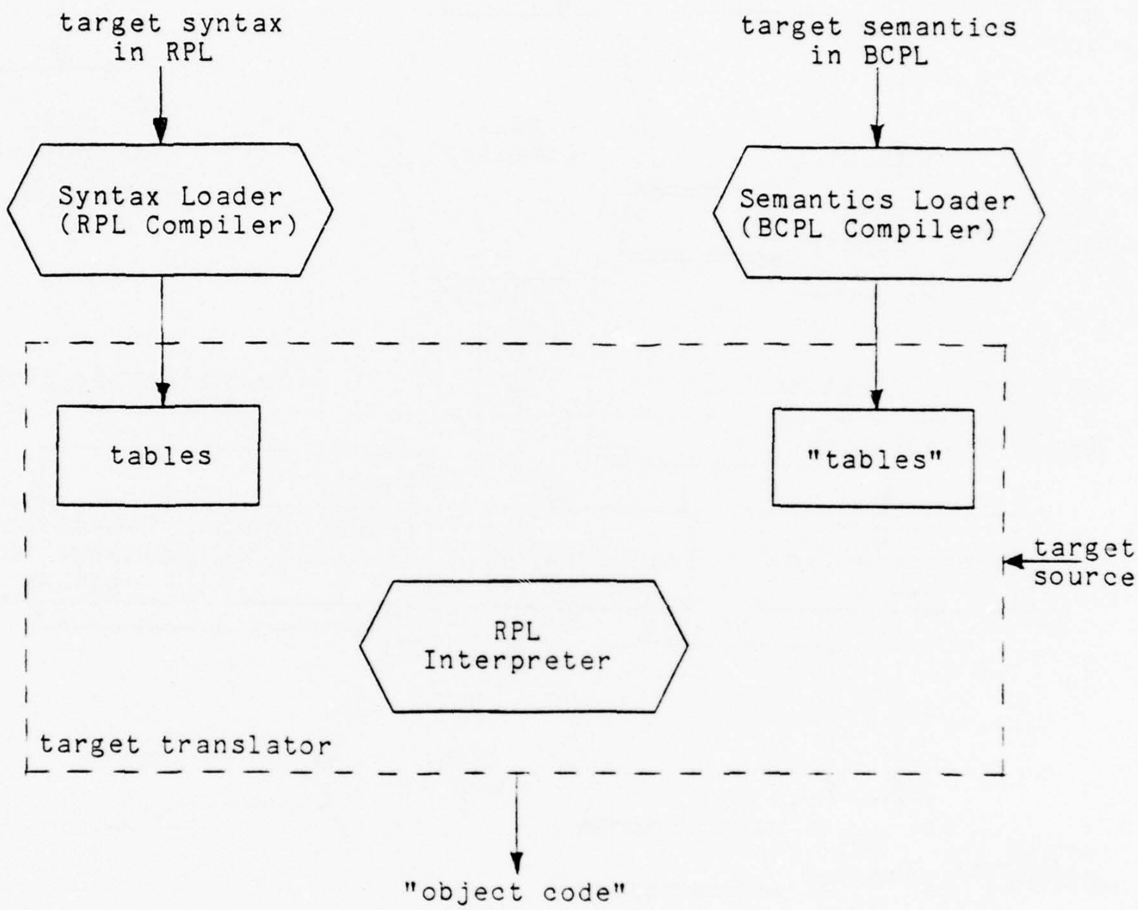


Figure 1. Basic Outline of the RPL System

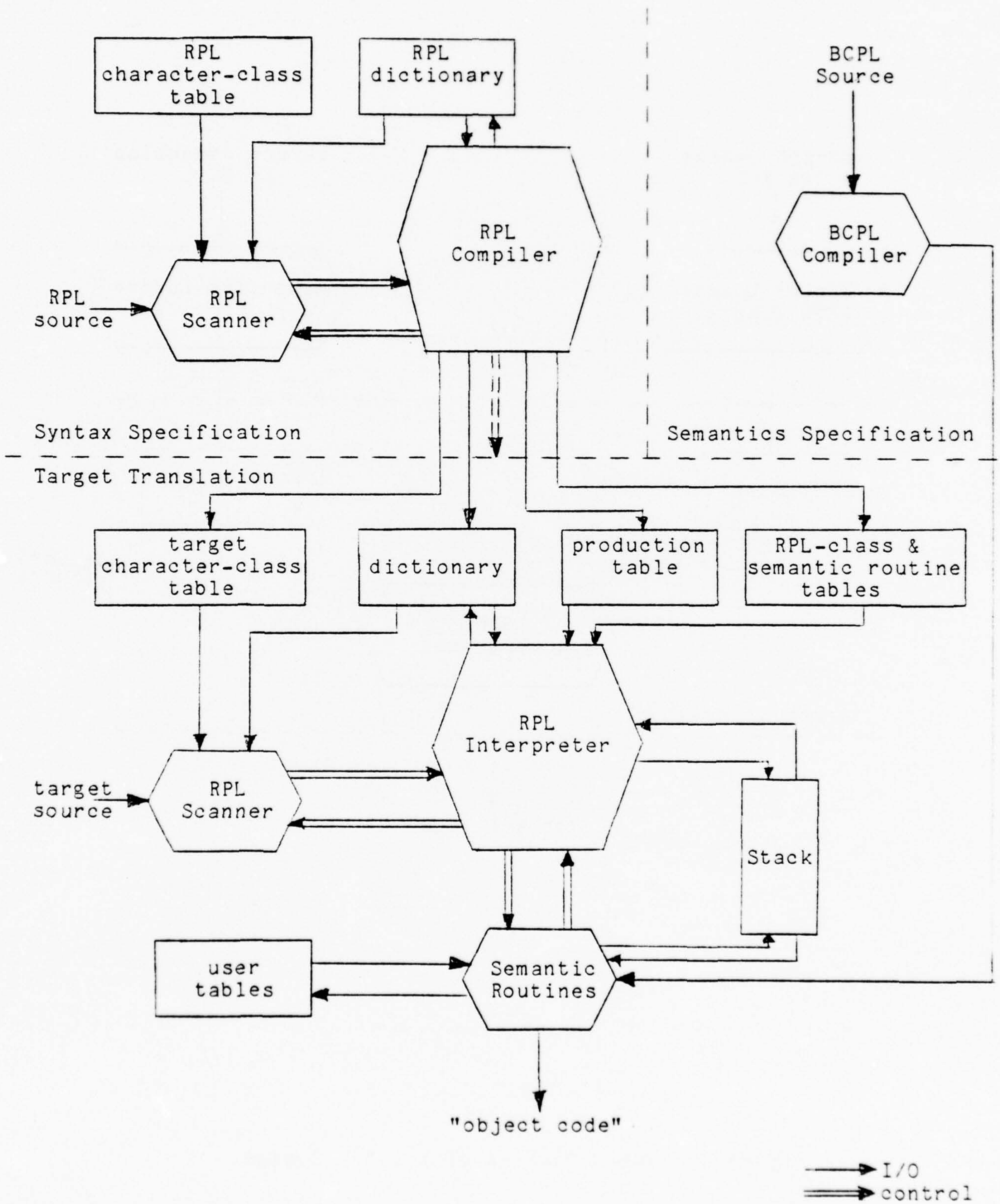


Figure 2. Detailed Flow of the RPL System

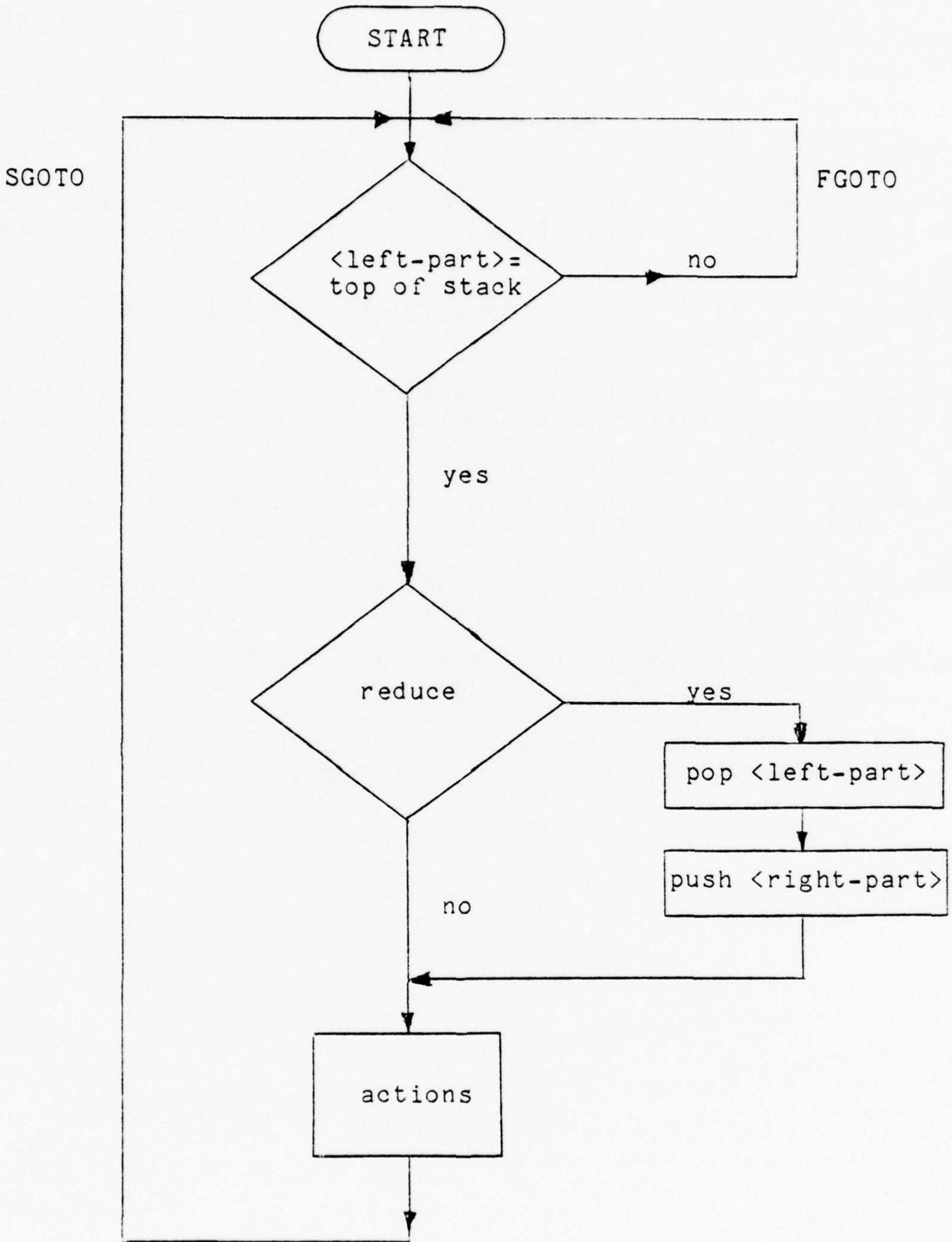


Figure 3. Production Language Interpretation

References

Curry, James E. BCPL Reference Manual. Computer Sciences Laboratory, Xerox Palo Alto Research Center. Palo Alto, Ca., 1975.

Evans, A. "An Algol60 Compiler," ACM National Conference, Denver, Colo., 1962.

Feldman, J. A. "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler," Comm. ACM 9 (Jan. 1966), 3-9.

Feldman, J. A. & D. Gries. "Translator Writing Systems," Comm. ACM 11 (Feb. 1968), 77-113.

Floyd, R. "A Descriptive Language for Symbol Manipulation," J. ACM 8 (Oct. 1961), 579-584.

Gries, D. Compiler Construction for Digital Computers. John Wiley & Sons, Inc. New York, 1971, Ch. 7.

Lantz, K. RPL User's Guide. Internal Memo, Computer Science Department, University of Rochester, 1977.

Appendix A. RPL Syntax

Note again that < and > are RPL source symbols, as well as BNF metasymbols. Their use is disambiguated by the fact that all RPL reserved words are upper case, whereas BNF nonterminals are lower case. Note also that this presentation is not intended to be optimal (in the sense of a minimal number of nonterminals); rather, it is intended to be as descriptive as possible.

```
<program>          ::= <RPL> <constructor> <declarator> <producer> <ENDRPL>

<constructor>      ::= <SCANNER>  {<scanner-decl>}*   <ENDSCANNER>

<scanner-decl>     ::= <char-class>
                       <delimiters>
                       <reserve>
                       <comments>

<char-class>       ::= <cc-name>  {<target-char>}*  <END>

<cc-name>          ::= DIGIT
                       HEAD
                       TRAIL
                       INVISIBLE
                       STRINGCONSTANT
                       CHARACTERCONSTANT
                       ESCAPE

<target-char>      ::= <printable-char>
                       <char-const>
                       <single-string>

<printable-char>::=  any printable ASCII character

<char-const>       ::= $<char>

<char>             ::= <printable-char>
                       *<escape-char>

<escape-char>      ::= as defined in section II.1.1

<single-string>    ::= "<string-char>"

<string-char>      ::= <char>  |  space

<delimiters>       ::= <DELIMITERS>  {<delimiter>}+   <END>

<delimiter>        ::= <single-dlm>
                       <double-dlm>

<single-dlm>       ::= <target-char>

<double-dlm>       ::= <printable-char><printable-char>
```

```
                          "<string-char><string-char>"

<reserve>           ::= <RESERVEDWORDS>  {<reserved-word>}+  <END>

<reserved-word>     ::= target identifier
                        <string-const>
                        <char-const>

<string-const>      ::= "{<string-char>}255"

<comments>          ::= <COMMENTS> {<tgt-symbol> <tgt-symbol>}+ <END>

<tgt-symbol>        ::= <delimiter>
                        <reserved-word>


<declarator>        ::= <DECLARATIONS>  {<RPL-decl>}*  <ENDDECLARATIONS>

<RPL-decl>          ::= <aliases>
                        <nonterminals>
                        <classes>
                        <constants>

<aliases>           ::= <ALIASES>  {<tgt-symbol> <alias>}+  <END>

<alias>             ::= <identifier>
                        <tgt-symbol>

<identifier>        ::= <head>{<trail>}254

<head>              ::= <letter>
                        @
                        <

<letter>            ::= A-Z  &  a-z

<trail>             ::= <letter>
                        <digit>
                        -
                        >

<digit>             ::= 0-9

<nonterminals>      ::= <NONTERMINALS>  {<nonterminal>}+  <END>

<nonterminal>       ::= <identifier>

<constants>         ::= <CONSTANTS>  {<constant> <integer>}+   <END>

<constant>          ::= <identifier>

<integer>           ::= <digit>{<digit>}4

<classes>           ::= <CLASSES>  {<class> {<class-member>+ }+ <END>
```

```
<class>             ::= <identifier>

<class-member>      ::= <tgt-symbol>
                        <alias>
                        <nonterminal>
                        <class>
                        <meta-symbol>
                        RESERVEDWORDS
                        DELIMITERS

<meta-symbol>       ::= I
                        INTEGER
                        STRINGCONST
                        CHARCONST


<producer>          ::= <PRODUCTIONS> {<production}+ <ENDPRODUCTIONS>

<production>        ::= {<label>:}* <left-part> [=><right-part>] {<action>}*

<label>             ::= <identifier>
                        START
                        SCANERROR
                        SWITCHERROR
                        SEMSWITCHERROR

<left-part>         ::= {<symbol>}+

<symbol>            ::= <RPL-symbol>[<index>]
                        <tgt-symbol>

<RPL-symbol>        ::= <class>
                        <nonterminal>
                        <basic-meta-symbol>
                        ANY

<index>             ::= 1 | 2 | 3

<right-part>        ::= {<symbol>}*

<action>            ::= <call>
                        <clean>
                        <error>
                        <exec>
                        <goto>
                        <halt>
                        <return>
                        <scan>
                        <semswitch>
                        <switch>

<call>              ::= CALL <label>
```

```
<clean>          ::= CLEAN

<error>          ::= ERROR <number>

<number>         ::= <integer>
                     <constant>

<exec>           ::= EXEC  {<proc-name> [(<parameter>)]}+

<proc-name>      ::= BCPL procedure name (identifier)

<parameter>      ::= <class>
                     <number>

<goto>           ::= {SGOTO  |  FGOTO} <label>

<halt>           ::= HALT <number>

<return>         ::= RETURN

<scan>           ::= SCAN [<number>]

<semswitch>      ::= SEMSWITCH {(<semcase-list> : <label>)}+

<semcase-list>   ::= <semcase> {, <semcase>}*

<semcase>        ::= <number>
                     DEFAULT

<switch>         ::= SWITCH <class> {(<case-list> : <label>)}+

<case-list>      ::= <case> {, <case>}*

<case>           ::= <class-member>
                     DEFAULT
```

Appendix B. SIMPLEGOL Syntax

```
<program>         ::= <expr>

<expr>            ::= <conditional>
                      <loop>
                      <assignment>
                      <case-expr>
                      <arithmetic>
                      <compound>
                      <control>

<conditional>     ::= IF <expr> THEN <expr> [ ELSE <expr> ] FI

<loop>            ::= [<prologue>] DO <expr> [<epilogue>] OD

<prologue>        ::= FOR <id> FROM <expr> [STEP <expr>] [<for-test><expr>]
                      <epilogue>

<id>              ::= <letter>{<trail>}*

<letter>          ::= A-Z

<trail>           ::= A-Z  |  0-9  |  _

<for-test>        ::= <boolean-test>
                      TO

<boolean-test>    ::= WHILE
                      UNTIL

<epilogue>        ::= <boolean-test> <expr>

<assignment>      ::= <variable> := <expr>

<variable>        ::= <id> [ (subscript-list) ]

<subscript-list> ::= <expr> {, <expr>}*

<case-expr>       ::= CASE <expr> OF <begin> <case-body> <end>

<begin>           ::= BEGIN [ <block-label> ]

<block-label>     ::= string constant

<end>             ::= END [ <block-label> ]

<case-body>       ::= <case> {; <case>}*

<case>            ::= <case-label-list> <expr>

<case-label-list>::= <case-label>: {<case-label>:}*
```

```
<case-label>        ::= <expr>  ...constant at compile-time...
                        DEFAULT

<arithmetic>        ::= <conjunction> {OR <conjunction>}*

<conjunction>       ::= <relational> {AND <relational}*

<relational>        ::= <term> {<rel-op> <term>}*

<term>              ::= <factor> {<add-op> <factor>}*

<factor>            ::= <primary> {<mult-op> <primary>}*

<primary>           ::= <constant>
                        <variable>
                        (<expr>)
                        <prefix-op> <primary>

<rel-op>            ::= =  ¦  ~=  ¦  <=  ¦  >=  ¦  <   ¦  >

<add-op>            ::= +  ¦  -

<mult-op>           ::= *  ¦  /

<prefix-op>         ::= -  ¦  ~

<constant>          ::= integer or real number

<compound>          ::= <begin> {<decl>;}* <body> <end>

<decl>              ::= <type> VECTOR <id-list> [ (<expr> : <expr>) ]
                        <type> <id-list>
                        <function>

<type>              ::= REAL
                        INTEGER

<id-list>           ::= <id> {, <id>}*

<function>          ::= [<f-type>] FUNCTION <id> ([<formals>]) := <expr>

<f-type>            ::= <type> [VECTOR]

<formals>           ::= <f-decl> {; <f-decl>}*

<f-decl>            ::= [<call-type>] <type> [ VECTOR ] <id-list>

<call-type>         ::= REFERENCE
                        VALUE

<body>              ::= expr {; <expr>}*

<control>           ::= <loop-ctl> [ OF <block-label> ] [ WITH <expr> ]
                        RETURN [ WITH <expr> ]
```

```
<loop-ctl>          ::= SKIPREST
                        BREAKOUT
```

Appendix C. XPRGOL -- RPL Source File

/***

Program: XPRGOL

Author: Keith A. Lantz   Date: February 24, 1977

Description: XPRGOL is an interactive line-oriented expression language. The only data type allowed is integer. Intermediary results may be stored in variables (with such assignments allowed wherever an integer or identifier may occur). Note that the logical operators interpret anything nonzero as true; zero means means false. The result of a logical expression is therefore 1 for true, 0 for false.

The user inputs an expression followed by either a carriage return, or the keyword "end"; the latter terminates an XPRGOL session. Unless the input expression is strictly an assignment, the result is output to the screen. A session might appear as follows:

```
? 2+2
   4
? a := 2^14-1+2^14
? a
   32767
? ~ b := 1
   0
? a - (b := b + 1)
   32765
? end
```

XPRGOL is formally defined as follows:

```
<session>        ::= {<expr>}* end

<expr>           ::= <conj> {<or-op> <conj>}*

<conj>           ::= <reln> {<and-op> <reln>}*

<reln>           ::= <term> {<rel-op> <term>}*

<term>           ::= <fact> {<add-op> <fact>}*

<fact>           ::= <exp> {<mult-op> <exp>}*

<exp>            ::= <prim> {^<prim>}*

<prim>           ::= (<expr>)
                     <integer>
                     <identifier> [:= <expr>]
                     <prefix-op> <prim>
```

```
        <or-op>          ::= or   ¦  '¦'
                             xor  ¦  %

        <and-op>         ::= and  ¦  &

        <rel-op>         ::= eq   ¦  =
                             ne   ¦  ~=
                             lt   ¦  <    ¦  ls
                             le   ¦  <=
                             gt   ¦  >    ¦  gr
                             ge   ¦  >=

        <add-op>         ::= +
                             -

        <mult-op>        ::= *
                             /
                             mod

        <prefix-op>      ::= not  ¦  ~
                             -


****************************************************************************/

<RPL>

<SCANNER>

        /* <HEAD> defaults to A-Z, a-z                                   */
        /* <DIGIT> defaults to 0-9                                       */
        /* <INVISIBLE> defaults to space and tab                         */
        /* no <CHARACTERCONSTANT>s or <ESCAPE>s                          */

    <TRAIL> A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
            a b c d e f g h i j k l m n o p q r s t u v w x y z
            0 1 2 3 4 5 6 7 8 9     <END>

    <STRINGCONSTANT>  <END> /* no <STRINGCONSTANT>s */

    <DELIMITERS> ( ) = < > ~ & ¦ % + - * / ^ $*C ~= <= >=  ::=     <END>

    <RESERVEDWORDS> end  mod
                    or  xor  and  not
                    eq  ne  lt  ls  le  gt  gr  ge  <END>

        /* no <COMMENTS>                                                 */

<ENDSCANNER>


<DECLARATIONS>
```

```
<NONTERMINALS>  <expr>  <conj>  <reln>  <term>  <fact>
                <exp>  <prim>   <END>

<ALIASES>
           /* operator equivalences      */

        ~          not
        &         and
        |         or
        %         xor
        =         eq
        <         lt                lt       ls
        >         gt                gt       gr
        ~=        ne
        <=        le
        >=        ge

           /* "pure" aliases -- for semantic routines and
              to avoid conflicts                          */

        :=        assign
        +         plus
        -         minus
        *         mult
        /         div
        ^         exp
        (         lparen
        )         rparen
        $*C       cret
<END>

<CLASSES>
           /* operator classes           */

        @orop              |  %
        @relop             =  ~=  <=   ">="  <  >
        @addop             +  -
        @multop            mult div mod
        @prefixop          not minus

           /* others                     */

        @fini              end $*C
        @primhead          @prefixop  (  I  INTEGER
<END>

<CONSTANTS>
        OK 0                                /* success code         */
        DIVERROR 1         MODERROR 2       /* results from Mult    */
        EXPERROR 1                          /* results from Exp     */
        NOTFOUND 1                          /* results from Stuff   */
        SUCCESS 1          FAILURE 2        /* HALT states          */
<END>
```

```
<ENDDECLARATIONS>

<PRODUCTIONS>

START:  #                                      EXEC Init SGOTO Session

    /* Provisions for run time error-handling.                  */

SCANERROR:  SWITCHERROR:  SEMSWITCHERROR:
Bug:    ANY                                    CLEAN


    /********************      <session>      ********************/

Session:
        #                                      EXEC Prompt SCAN ^
                                                 SGOTO StackOK
        ANY                                    ERROR 1 SGOTO Bug

    /* Watch out for leading terminators -- carriage returns and "end"s.*/

StackOK:@fini =>                               SWITCH @fini ^
                                                 (cret: Session) ^
                                                 (end: Halt)

        ANY                                    CALL Expression
        <expr> @fini =>                        EXEC Print ^
                                               SWITCH @fini ^
                                                 ($*C: Session) ^
                                                 (end: Halt)

        ANY                                    ERROR 2 SGOTO Bug

    /********************      <expr>         ********************/

Expression:
        ANY                                    CALL Conjunction
        <conj> @orop <conj> ANY => <conj> ANY  EXEC Or(@orop)
        <conj> @orop                           SCAN SGOTO Expression
        <conj> ANY => <expr> ANY               RETURN

        ANY                                    ERROR 3 SGOTO Bug

    /********************      <conj>         ********************/

Conjunction:
        ANY                                    CALL Relational
        <reln> & <reln> ANY => <reln> ANY      EXEC And
        <reln> and                             SCAN SGOTO Conjunction
        <reln> ANY => <conj> ANY               RETURN
```

```
        ANY                                     ERROR 4 SGOTO Bug

    /*******************        <reln>          *******************/

Relational:
        ANY                                     CALL Term
        <term> @relop <term> ANY => <term> ANY  EXEC Rel(@relop)
        <term> @relop                           SCAN SGOTO Relational
        <term> ANY => <reln> ANY                RETURN

        ANY                                     ERROR 5 SGOTO Bug

    /*******************        <term>          *******************/

Term:
        ANY                                     CALL Factor
        <fact> @addop <fact> ANY => <fact> ANY  EXEC Add(@addop)
        <fact> @addop                           SCAN SGOTO Term
        <fact> ANY => <term> ANY                RETURN

        ANY                                     ERROR 6 SGOTO Bug

    /*******************        <fact>          *******************/

Factor:
        ANY                                     CALL Exponential
        <exp> @multop <exp> ANY => <exp> ANY    EXEC Mult(@multop)^
                                                SEMSWITCH ^
                                                  (0: FactOK) ^
                                                  (DIVERROR: DivBug) ^
                                                  (MODERROR: ModBug)
FactOK: <exp> @multop                           SCAN SGOTO Factor
        <exp> ANY => <fact> ANY                 RETURN

        ANY                                     ERROR 7 SGOTO Bug
DivBug: ANY                                     ERROR 8 SGOTO Bug
ModBug: ANY                                     ERROR 9 SGOTO Bug

    /*******************        <exp>           *******************/

Exponential:
        ANY                                     CALL Primary
        <prim> $^ <prim> ANY => <prim> ANY      EXEC Exp ^
                                                SEMSWITCH ^
                                                  (OK: ExpOK) ^
                                                  (EXPERROR: ExpBug)
ExpOK:  <prim> exp                              SCAN SGOTO Exponential
        <prim> ANY => <exp> ANY                 RETURN

        ANY                                     ERROR 10 SGOTO Bug
```

```
    ExpBug: ANY                                          ERROR 11 SGOTO Bug

        /********************          <prim>          ********************/

    Primary:
            @primhead                                    SCAN ^
                                                         SWITCH @primhead ^
                                                           (INTEGER: HaveInt) ^
                                                           (I: HaveIdent) ^
                                                           ($(: Nested) ^
                                                           (DEFAULT: Prefixed)

            ANY                                          ERROR 12 SGOTO Bug

        /*          <prim> ::= <integer>                                  */

    HaveInt:
            INTEGER ANY => <prim> ANY                    SCAN RETURN

        /*          <prim> ::= <identifier> [:= <expr>]                   */

    HaveIdent:
            assign                                       EXEC CheckLevel ^
                                                           SCAN SGOTO Assignment
            I ANY => <prim> ANY                          EXEC Stuff ^
                                                         SEMSWITCH ^
                                                           (OK: IOK) ^
                                                           (NOTFOUND: IBug)
    IOK:    ANY                                          RETURN

    IBug:   ANY                                          ERROR 13 SGOTO Bug

    Assignment:
            ANY                                          CALL Expression
            I assign <expr> ANY => <prim> ANY            EXEC Assign RETURN

            ANY                                          ERROR 14 SGOTO Bug

        /*          <prim> ::= (<expr>)                                   */

    Nested:
            ANY                                          CALL Expression
            ( <expr> )  => <prim>                        EXEC Move SCAN RETURN

            ANY                                          ERROR 15 SGOTO Bug

        /*          <prim> ::= <prefix-op> <prim>                    */

    Prefixed:
            ANY                                          CALL Primary
            @prefixop <prim> ANY => <prim> ANY           EXEC Prefix(@prefixop) ^
                                                           RETURN
```

```
ANY                                              ERROR 16 SGOTO Bug

    /********************      Halt state(s)     ********************/

Halt:   #                                        HALT SUCCESS
        ANY                                      HALT FAILURE

<ENDPRODUCTIONS>

<ENDRPL>
```

Appendix D. XPRGOL -- Semantic Routines

```
// XprSemantics.Bcpl -- Semantic Routines for XPRGOL
// KAL -- 2/24/77

get "Xpr.Head"                   // RPL-Compiler-generated header file
get "RPLInterpreter.Head"        // RPL System header
get "MyXpr.Head"                 // XPRGOL header

static  IsAssignment

manifest
  [
        TRUE    = 1
        FALSE   = 0
  ]


        // Init -- Initialize the world

let Init () be
  [
     InitSymbolTable ()
  ]


        // Prompt -- Issue a prompt for the next input expression

and Prompt () be
  [
     Ws ("*C? ")
     IsAssignment = false        // don't know yet if we have an assignment
     TokenCount = 0              // global count kept by scanner
  ]


        // Print the answer if the line wasn't an assignment

and Print () be
  [
     unless IsAssignment do
       [
          Ws ("   ")
          Wns (dsp, L2)
       ]
  ]


        // Or -- x OR y  = FALSE iff x=y=0
        //                 TRUE otherwise
        //       x XOR y = FALSE iff x=y=0 or x and y are non-zero
        //                 TRUE otherwise
```

```
    and Or (tkn) be
      [
         let success = selecton tkn into
           [
           case OR: (L4 ne 0) % (L2 ne 0)
           case XOR:((L4 eq 0) & (L2 ne 0)) % ((L4 ne 0) & (L2 eq 0))
           ]
         R2 = success ? TRUE, FALSE
      ]


            // And -- x AND y = TRUE iff x is non-zero and y is non-zero
            //                  FALSE otherwise

    and And () be
      [
         R2 = (L4 ne 0) & (L2 ne 0) ? TRUE, FALSE
      ]


            // Rel -- x EQ y = TRUE iff x = y
            //                 FALSE otherwise
            //        x NE y = FALSE iff x = y
            //                 TRUE otherwise
            //        x LT y = TRUE iff x < y
            //                 FALSE otherwise
            //        x GT y = TRUE iff x > y
            //                 FALSE otherwise
            //        x LE y = TRUE iff x <= y
            //                 FALSE otherwise
            //        x GE y = TRUE iff x >= y
            //                 FALSE otherwise

    and Rel (tkn) be
      [
         let success = selecton tkn into
           [
           case EQ: L4 eq L2
           case NE: L4 ne L2
           case LT: L4 ls L2
           case LE: L4 le L2
           case GT: L4 gr L2
           case GE: L4 ge L2
           ]
         R2 = success ? TRUE, FALSE
      ]


            // Add -- x PLUS  y = x + y
            //        x MINUS y = x - y

    and Add (tkn) be
      [
         R2 = selecton tkn into
```

```
          [
          case PLUS:        L4 + L2
          case MINUS:       L4 - L2
          ]
      ]


          // Mult -- x MULT y = x * y
          //         x DIV y  = x / y
          //         x MOD y  = x rem y

    and Mult (tkn) be
      [
        SemSwitch = OK
        switchon tkn into
          [
          case DIV:

               test L2 eq 0
                 ifso  SemSwitch = DIVERROR
                 ifnot R2 = L4 / L2

          endcase

          case MULT:

               R2 = L4 * L2

          endcase

          case MOD:

               test L2 < 0
                 ifso  SemSwitch = MODERROR
                 ifnot R2 = L4 rem L2

          endcase
          ]
      ]


          // Exp -- x EXP y = x to the y-th power

    and Exp () be
      [
        SemSwitch = OK
        test L2 < 0
          ifso  SemSwitch = EXPERROR
          ifnot
            [
              let temp = 1
              for i = 1 to L2 do
                   temp = temp * L4
              R2 = temp
```

```
            ]
    ]


            // CheckLevel -- Check level of assignment to determine whether
            //      the entire input expression is strictly an assignment
            //      (e.g., "?a:=2") -- i.e., whether ":=" is the second
            //      token parsed

    and CheckLevel () be
      [
        if (TokenCount eq 2) then IsAssignment = true
      ]


            // Stuff -- Stuff value of identifier onto the stack

    and Stuff () be
      [
        let handle = LookUpInSymbolTable (L2)
        if (handle eq NULL) do      // identifier is not defined
          [
            SemSwitch = NOTFOUND
            return
          ]
        R2 = handle>>Symbol.Value  // stuff variable's value into <primary>
        SemSwitch = OK
      ]


            // Assign -- Assignment

    and Assign () be
      [
            // If this is the first occurrence of the identifier, initialize
            // a symbol table entry for it.

        let handle = LookUpInSymbolTable (L4)
        if (handle eq NULL) then handle = NewSymbol (L4)
        handle>>Symbol.Value = L2  // set value of variable to <expr>
      ]


            // Prefix -- NOT x      = TRUE iff x=0
            //                        FALSE otherwise
            //           MINUS x    = -x

    and Prefix (tkn) be
      [
        R2 = selecton tkn into
          [
          case NOT:        (L2 eq 0) ? TRUE, FALSE
          case MINUS:      -L2
          ]
```

```
    ]


            // Move -- reduce a nested expression to a primary

  and Move () be
    [
       R1 = L2
    ]



  and UserError (n) be
    [
       Ws ("   ********** USER ERROR : ");  Wns (dsp, n)
    ]


  and UserHalt (n) be
    [
       Ws ("   ********** HALT : ");  Wns (dsp, n)
       Ws ("*CType any key to finish....")
       Gets (keys)
    ]
```

Appendix E. XPRGOL -- Compiler-Generated Header File

```
// Xpr.Head -- Externals and manifests

external            // semantic routines
    [
        Init
        Prompt
        Print
        Or
        And
        Rel
        Add
        Mult
        Exp
        CheckLevel
        Stuff
        Assign
        Move
        Prefix
    ]

manifest            // class tokens
    [
        EQ                       = 3      // "=" = "eq"
        LT                       = 4      // "<" = "lt" = "ls"
        GT                       = 5      // ">" = "gt" = "gr"
        NOT                      = 6      // "~" = "not"
        AND                      = 7      // "&" = "and"
        OR                       = 8      // "¦" = "or"
        XOR                      = 9      // "%" = "xor"
        PLUS                     = 10     // "+" = "plus"
        MINUS                    = 11     // "-" = "minus"
        MULT                     = 12     // "**" = "mult"
        DIV                      = 13     // "/" = "div"
        EXP                      = 14     // "^" = "exp"
        NE                       = 16     // "~=" = "ne"
        LE                       = 17     // "<=" = "le"
        GE                       = 18     // ">=" = "ge"
        MOD                      = 20     // "mod"
    ]

manifest            // constants
    [
        OK                       = 0
        DIVERROR                 = 1
        MODERROR                 = 2
        EXPERROR                 = 1
        NOTFOUND                 = 1
        SUCCESS                  = 1
        FAILURE                  = 2
    ]
```